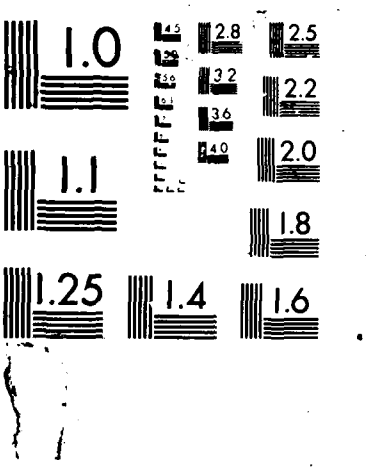
AD-A189 200 DESCRIBING CONSTRAINTS ON A DIGITAL CIRCUIT'S BEHAVIOR 1/1
(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE
D BALDWIN JUL 87 TR-222 DACA76-85-C-0001

UNCLASSIFIED F/G 9/1 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

Describing Constraints on a Digital Circuit's Behavior

Doug Baldwin
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 222
July 1987

Abstract

Department of Computer Science
University of Rochester
Rochester, New York 14627

Describing Constraints on a Digital Circuit's Behavior

Doug Baldwin
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 222
July 1987

Abstract

Automatic design of circuits from high-level descriptions of their behavior requires that some physical constraints be included in behavioral specifications. This paper describes a simple but powerful mechanism for doing so. The key ideas behind this approach are *attributes* to represent physical parameters of a circuit and *constraint statements* to restrict the values that attributes may assume. Practical circuits have been described using these ideas. An algorithm for extracting constraints from specifications and deciding which parts of the specifications are subject to which constraints is also presented and proven correct.

This report is based on work funded in part by the National Science Foundation under grants DMC-8613489 and DRC-8320136, and by the U.S. Army Engineering Topographic Laboratories under research contract number DACA76-85-C-0001. The views stated in this report are those of the author, and do not necessarily reflect those of any of the funding agencies.

We thank the Xerox Corporation University Grants Program for providing equipment used in preparation for this paper.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 222	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Describing Constraints on a Digital Circuit's Behavior		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Douglas Baldwin		8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Computer Science 535 Computer Studies Bldg. Univ. of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA / 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE July 1987 13. NUMBER OF PAGES 26
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* unclassified 15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	
A-1	

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

circuit description languages
circuit synthesis
design

DTIC COPY INSPECTED

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Automatic design of circuits from high-level descriptions of their behavior requires that some physical constraints be included in behavioral specifications. This paper describes a simple but powerful mechanism for doing so. The key ideas behind this approach are attributes to represent physical parameters of a circuit and constraint statements to restrict the values that attributes may assume. Practical circuits have been described using these ideas. An algorithm for extracting constraints from specifications and deciding which parts of the specifications are subject to which constraints is also presented and proven correct.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Contents

1 Introduction

Behavioral synthesis can be loosely defined as the process of translating a high-level description of what one wants a circuit to do into a circuit design ready for fabrication. General definitions of "high-level" and "ready for fabrication" are hard to come by, but for the purposes of this paper "high level" means about as abstract as a modern programming language, and "ready for fabrication" means at least as detailed as a gate or gate-macro level schematic diagram. People have been working for a long time on computer programs to automate behavioral synthesis, but only recently have they had any success. The difficulty of behavioral synthesis is hard to understand if the above definition is accepted, since Figure 1 seems to present a satisfactory solution: a conventional high-level language compiler's output is programmed into a read-only memory, which is then embedded in an otherwise fixed microprocessor design to form the synthesized "hardware". Of course this solution is not satisfactory, and the reason is that the definition of behavioral synthesis ignored a very important point: not only must the synthesized circuit exhibit the same behavior as the input specification, it must also do so within certain cost limits. These limits may be expressed as bounds on acceptable power consumption, layout area, speed, testability, et cetera. It is the need to design subject to these *constraints* that changes behavioral synthesis from a trivial application of well-known compiler technology into the extremely difficult problem that it really is.

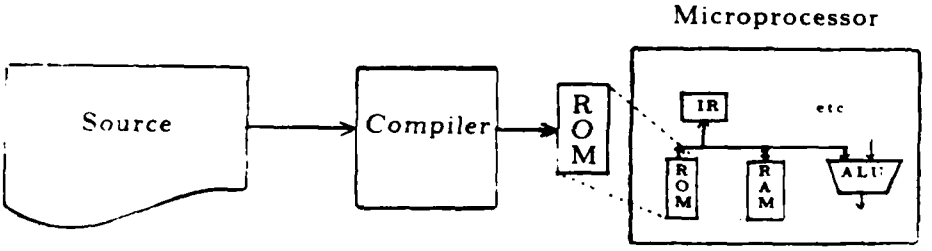


Figure 1. A Naive Circuit Synthesizer

Constraints on circuit design also arise from purely behavioral considerations. In other words, there are cases in which a circuit cannot be said to behave correctly if it does not meet certain physical constraints. This follows from the fact that circuits are not used in isolation, but rather in conjunction with other devices (e.g., sensors and displays that provide inputs and present outputs). In order to properly interface with these devices, a circuit must transmit and receive data at certain speeds, encode it as currents or voltages according to some agreed-upon standard, et cetera. Just as cost requirements do, these interface protocols define bounds on circuit speed, signal levels, et cetera.

Constraints can be either static or dynamic. A constraint is static if it does not change as the circuit does different things. A good example is a limit on layout area. Dynamic constraints can change as a circuit runs. Examples include devices with variable data transmission or reception rates, and devices with a low-power "stand-by" mode. As a general rule of thumb, the more intimately a constraint is connected

to a circuit's behavior the more dynamic it is likely to be, while constraints that limit the costs of building or operating a circuit tend to be static.

This paper describes a simple way of combining the specification of constraints with that of more conventional forms of behavior in a single language. The scheme is based on the ideas of *attributes* that represent physical quantities associated with variables or code fragments, and *constraint statements* that describe constraints in terms of these attributes. This scheme is quite flexible despite its simplicity. With only attributes and constraint statements as basic notions it describes both those constraints that are inherent parts of a circuit's behavior and those that are optimization goals for its implementation. Similarly, it handles both static and dynamic constraints. Constraints are vital parts of a complete circuit description, and so are described in terms that are consistent with an otherwise high level of abstraction. For example, resource constraints are given in direct terms of commonly used quality measures like power consumption or layout area rather than in indirect terms like number of parts, timing is described in terms of actions in the behavioral description rather than device speeds, et cetera. The basic approach is easy to integrate into larger behavioral description languages. This has been demonstrated for two languages, one that was not originally designed with constraints in mind and one that was. The utility of the scheme has been demonstrated in descriptions of several realistic circuits.

This work is novel for a number of reasons:

- It offers a single notation that can describe constraints used for several distinct purposes (for example, static constraints used to optimize a circuit's implementation versus dynamic constraints that are integral parts of its behavior). Furthermore, this notation fits comfortably into a highly abstract (relative to the final hardware) language. Much previous work on constraints in circuit synthesis considered only static constraints for optimization, and required them to be given to synthesizers independently of other aspects of the circuit's behavior (examples include RT-CAD [16], Hafer's work [9], and quite recently Sehwa [14]). Mimola [12] is an example of a system that does not separate constraint specification from other aspects of behavior, but does require constraints to be described at a lower level of abstraction than everything else. Mimola's constraints are given in terms of hardware elements (e.g., the number of modules of a given type available, their speeds, et cetera), whereas the rest of its input consists of high-level algorithms.
- It is able to deal with dynamic constraints as well as static ones. Several recent reports [6, 13] have described ways of including dynamic timing constraints in behavioral descriptions. That work is roughly equivalent to the "time points" (see below) of mine, but does not include the general system of attributes and constraint statements that I do.
- It deals with the need to understand constraints at compile time rather than run time. The idea of attributes to represent physical features of a circuit has already appeared in VHDL [10] and ISPS [3], and both VHDL and CONLAN [15] allow programmers to write "assertions" that are superficially similar

to constraint statements. However, constraint statements are intended to be "evaluated" at *compile* time in order that a synthesizer can take the necessary actions to ensure that the corresponding condition holds in the final circuit. whereas an assertion is intended to be evaluated at *run* time (or simulation time), and its success or failure is simply a check that a simulation is running as expected. The assertion facilities in languages like VHDL or CONLAN are too general to guarantee that assertions can be understood at compile time. The system described here has the restrictions needed for compile time handling of constraints, yet appears to be expressive enough for use in serious circuit design.

2 Describing Constraints

The basic idea is to associate *attributes* with parts of a specification to represent physical quantities associated with those parts. For example, a procedure or other block of code might have attributes for the power consumption or layout area of its final implementation, a variable might have attributes describing how the values it takes on are encoded as voltages, et cetera. A constraint is simply a statement that restricts the values that an attribute can have. For example, constraints might instruct a synthesizer to minimize certain areas or power consumptions, use certain voltage ranges to encode a variable's values, et cetera. Because constraints are introduced by executable statements, they can be either static or dynamic depending on whether the appropriate statements are executed once or more than once. These ideas define a schema according to which any number of different languages can be built. They are presented by way of the following representative example and a discussion of their uses in several real languages, with the reader expected to adapt them in whatever way seems most useful for specific situations.

Figure 2 is an example of how attributes and their constraints can be used in a behavioral specification. The example illustrates the key features of this approach, but does not demonstrate all possible attributes or constraint types. The language is Pascal-like, but is not intentionally based on any existing behavioral description language. Lines are numbered for later reference. The circuit described by the example is a simple serial transmitter. Its behavior has been simplified as much as possible while still demonstrating significant uses of constraints. The transmitter is supposed to send data at 9600 bits per second, using a voltage of $-12 \pm .5$ Volts to transmit a logic 1 and $+12 \pm .5$ Volts to transmit a 0. The host places characters to transmit on the transmitter's "char" input, then asserts the "start" signal to begin transmission. Every character transmitted is preceded by a start bit (a 0) and is followed by at least 2 stop bits (1's). An "abort" input is provided with which the host can halt the transmitter in the middle of a transmission, forcing it to begin

```
(1)  circuit xmitter
(2)     input char 8 bits
(3)     input start 1 bit
(4)     input abort 1 bit
(5)     output xmit 1 bit
(6)     local i 3 bits
(7)     xmit'logic_0 :≤ 12.5 Volts & xmit'logic_0 :≥ 11.5 Volts
(8)     xmit'logic_1 :≤ −11.5 Volts & xmit'logic_1 :≥ −12.5 Volts
(9)     minimize xmitter'power
(10)    loop
(11)      if start
(12)        xmit'hold_time := 104 microseconds
(13)        i := 0
(14)        do
(15)           timepoint started
(16)           xmit := char bit i
(17)           i := i + 1
(18)        while i ≠ 0 and not abort
(19)        xmit'hold_time :≥ 208 microseconds
(20)        xmit := 1
(21)        started'elapsed_time :≤ 200 microseconds
(22)      else
(23)        unconstrain xmit'hold_time
(24)        xmit := 1
(25)      end if
(26)    end loop
(27) end circuit
```

Figure 2. A Serial Transmitter Described with Constraints

sending stop bits. Aborts must be recognized and acted upon within 200 microseconds of the "abort" input being asserted. The important points demonstrated by the example are as follows:

Attributes. Attributes are identified in this paper by a VHDL-style "*object'attribute*" notation that indicates both the attribute and the part of the specification to which it pertains. Attributes are associated with variables, blocks of code, and special objects called time points (discussed later). Generally, different sets of attributes are relevant to different kinds of object. The attributes used in the example, the lines at which they are first mentioned, and their meanings are as follows:

"Logic_0" (7). The voltage that encodes a logical 0. Associated with variables.

"Logic_1" (8). The voltage that encodes a logical 1. Associated with variables.
"Power" (9). The power consumption of the circuit built to implement a block of code. Associated with code blocks.
"Hold_Time" (12). The time between assignments to a variable. Associated with variables.
"Elapsed_Time" (21). The time since a time point was last executed. Associated with time points.

The "hold_time" and "elapsed_time" attributes are discussed in more detail below. Note that this selection of attributes is representative (but not exhaustive) of the kinds of attribute I have found useful. It is not the final word on possible attributes, nor should all of the attributes mentioned here be relevant in every synthesizer.

Having attributes associated with blocks of code raises the question of what exactly is a block. In order for attributes like power consumption or layout area to be meaningful for a block, the block must correspond to an identifiable section of the final circuit. Thus the definition of "block" really depends on the level of granularity at which a synthesizer will try to optimize separate elements of a specification into single pieces of hardware. In some cases attributes of individual expressions or statements might be meaningful, in others the only meaningful attributes might be those of large sub-programs or even the entire specification. In the example, the only attribute of code blocks that is used is "xmitter'power", representing the power consumption of the entire circuit.

Timing. Timing constraints are probably among the most important in describing a circuit's behavior, and so I propose several different mechanisms for dealing with them. Many timing constraints serve to set the rate at which data are transmitted to some output or received from some input. The "hold_time" attribute defined above is convenient for setting transmission rates with a single constraint. Examples of the use of "hold_time" can be seen in lines 12 and 19 of the example, where the time between consecutive assignments to "xmit" is forced to be 104 microseconds for the 9600 bits per second data transmission, and 208 microseconds for the two bit times worth of ones between characters. A complementary attribute ("ignore_time") can be defined as the time between reads of a variable. Constraints on "ignore_time" are typically used to limit the rate at which inputs to a circuit are tested.

Although "hold_time" and "ignore_time" can be used to describe most timing constraints, there are cases in which one wants to constrain the time between events other than reads or writes of a variable. "Time points" are introduced to describe this more general kind of timing. A time point is a marker that can be placed anywhere in the executable part of a specification. Every time point has an "elapsed_time" attribute that represents the amount of time since control last passed the time point. In the example, a time point called "started" is placed at the beginning of the "do" loop (line 15), thus capturing the time at which each iteration begins. Line 21 constrains the time between the beginning of the loop (and hence the latest test of "abort") and the beginning of the first stop bit. This

constraint reflects the requirement that aborts be recognized and acted on within 200 microseconds of being asserted.

Basic Constraints. The constraints used in the example are equality (":="), less than or equal (":≤"), greater than or equal (":≥") and "minimize" (see lines 7, 8, 9, 12, 19, and 21). As with attributes, other kinds of constraint can easily be imagined. The notation used for equality and the two inequalities deliberately reflects their role as executable "assignments", but they differ from conventional assignments in an important way: The actual value of an attribute is determined by the circuit generated from a specification. Therefore, what users really do with operators like ":≤" is place constraints on implementations, hopefully using terms that are meaningful in abstract, high-level, specifications. It is then the ***synthesizer*** that must understand what these operators mean and how its actions will affect attribute values if acceptable implementations are to be produced. Thus any assignment implied by a constraint operator happens rather indirectly, and primarily at compile time rather than during circuit operation. The deep understanding that synthesizers must have of constraint operators and attributes means that it is probably not practical to allow user-defined attributes in my system — it is too difficult to tell synthesizers what they mean.

Dynamic constraints. The transmitter's specification requires both static and dynamic constraints. The signal levels for "xmit" and the goal for power consumption do not depend on what the transmitter is doing, and so are static constraints. The relevant attributes are constrained once (lines 7 through 9) and never changed. On the other hand, the hold time for "xmit" does vary depending on what the transmitter is doing — while transmitting a character it must be one bit time (104 microseconds), after each character it must be two bit times, and when transmissions are not even requested ("start" inactive) it can be anything at all. These dynamic constraints are described by lines 12, 19, and 23 in the example. Each of these lines defines a section of the specification in which a distinct constraint applies to "xmit'hold_time". Note in particular line 23, which demonstrates an "unconstrain" statement that removes all constraints from an attribute.

Simultaneous constraints. One feature that the expression of constraints does share with normal assignments is that an attribute can only have one constraint at a time. This requirement prohibits many important uses of constraints, and so must be circumvented somehow. The serial transmitter contains a good example of the need for multiple simultaneous constraints in the requirement that transmitted ones and zeros be represented by voltages within certain ranges. I use "&" as a way of combining simple constraints into more complicated ones. Thus the ranges mentioned above are described on lines 7 and 8 by stating that the voltage representing a logic value is above a lower bound and at the same time below an upper bound.

Default constraints. Specifications will be easier to write if attributes can have default constraints. For instance, Figure 2 does not state the voltage levels used to encode logic values on the transmitter's inputs. A typical default might be that unless otherwise specified logic 0's are represented by 0.5 ± 0.5 Volts and logic 1's

by 4.5 ± 0.5 Volts, in which case the transmitter specification is exactly the same as if it had included lines of the form

$$start'logic\_0 :\leq 1 \textbf{ Volt } \& \ start'logic\_0 :\geq 0 \textbf{ Volts}$$

and so forth. In other cases a good default is to have no constraint on an attribute. An example is the "ignore_time" of the transmitter's inputs. Circuit speed, including the rate at which these inputs are tested, is adequately controlled by the constraints on "xmit'hold_time" and "started'elapsed_time", and so additional timing constraints are unnecessary.

Complexity of constraints. All of the constraints used in this example have been very simple, being either minimizations of a single attribute or comparisons between an attribute and a constant. Although simple constraints of this sort seem adequate for practical circuit design (see Section 4), uses can be found for more elaborate ones. For instance, one might want to limit some combination of attributes (speed-power product is a good example), or constrain one combination of attributes relative to another. Extensions to my constraint definition syntax to allow more general constraints are straightforward. However, one must keep in mind that a synthesizer has to be able to tell fairly quickly what attribute values solve a constraint and how to fix things if a constraint is violated (i.e., which attributes should have higher values, which lower, et cetera). Constraint satisfaction quickly becomes very expensive (NP-Hard or worse) as more elaborate forms of constraint are allowed. Thus, while more complicated constraints than the ones I use can be written, the complexity of solving them and their apparently infrequent use discourage doing so. Those who nonetheless feel a need for constraints more elaborate than the ones discussed in this paper can find a number of satisfaction heuristics in constraint-based programs developed for other applications [e.g., 5, 8, 17].

The most important reason for wanting more elaborate constraints than the ones I propose is to describe trade-offs between attributes exactly. For example, suppose the following two statements appear in a specification:

minimize *some_object'power*
maximize *some_object'speed*

These two requirements are probably contradictory, in that increasing circuit speed generally requires increasing power consumption. The user would probably find it much more useful to describe the ways in which he or she is willing to trade power consumption for speed. Partial but nonetheless useful descriptions of such trade-offs can be written by combining minimization or maximization with an upper or lower bound. For example, replacing the above requirements with

minimize *some_object'power* & *some_object'power* $:\leq 100$ **milliWatts**
maximize *some_object'speed* & *some_object'speed* $:\geq 10$ **megaHertz**

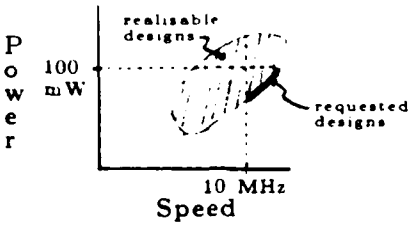


Figure 3. Cooperating Speed and Power Constraints

absolutely forbids trading power for speed if doing so yields a power consumption greater than 100 milliWatts. Similarly, power reductions that require a speed below 10 megaHertz are not acceptable. Within these limits the trade-off of speed and power is undefined, although the synthesizer should generally try to increase speed and decrease power consumption simultaneously. Figure 3 shows how these constraints restrict designs to a particular part of the border of the space of possibilities.

3 Implementation

So far I have described a very simple set of attributes and constraints that allows physical aspects of circuit behavior to be stated in behavioral descriptions. Because one hopes to have automatic translation of behavioral descriptions into circuit designs some day, it is also important to consider the implications of constrainable attributes for such translation. Obviously the full impact of constraints will be understood only through the development of constraint-sensitive translators. One such project already under way is the RASP system[1] [2] at the University of Rochester. Although this system is still in the early stages of design, some of the issues involved in dealing with constraints have become clear. This section discusses these issues and the ways in which they will be addressed in RASP.

3.1 Overview

Very roughly speaking, there are two parts to the problem of generating a circuit from a specification with constraints. The obvious one is finding a translation from the specification to a circuit design that meets the constraints. The less obvious one is figuring out the constrained *intervals* in the specification, i.e., figuring out which groups of source statements are subject to which constraints. Since constraints are allowed to be assigned to attributes, constrained intervals must be found through some form of flow analysis. Fortunately this analysis is quite similar to that used in conventional compilers [1], and so is fairly easy to implement. A general algorithm for finding constrained intervals is discussed in the next section.

In RASP, timing constraints are the only ones that are dynamic, and so circuit generation will largely be driven by them. The intervals arising from timing constraints will be identified by a version of the algorithm described in the next section.

[1] Originally called ASP, the name was changed to avoid confusion with another project.

Each interval will then be translated into a sequence of circuit control steps. Timing constraints will be met by varying the extent to which sequential source operations are compacted into parallel circuit operations. The extent to which compaction can be done without changing the meaning of the specification is obviously limited by data dependencies between source statements, but it is also limited by time points and "elapsed_time" constraints. Specifically, the notion that control passes a time point or "elapsed_time" constraint at a well-defined time prevents code from moving arbitrarily past these forms. The easiest solution is simply to disallow any code motions across these constructs. This solution has the drawback that time points and uses of "elapsed_time" become possibly unnecessary barriers to parallelism. More sophisticated analysis in the translator could probably identify certain kinds of code that can move across time points and related constraints, thus reducing this barrier. Because a parallel circuit generally contains more hardware than a serial one, compaction is also limited by constraints on power consumption and layout area. RASP's compactor will be based on microcode compaction heuristics [7, 11], modified to take into account the more complicated interactions between constraints. Intervals will be compacted one at a time, in order from shortest (i.e., least time allowed) to longest. This ordering is a heuristic to deal with the problem of overlapping intervals: doing more tightly constrained intervals early reduces the chances of being unable to meet constraints on later ones because of poor scheduling of the first.

3.2 An Algorithm for Identifying Constrained Intervals

Constrained intervals are easily identified in a flow graph of a behavioral specification. A form of depth-first traversal is used to explore all possible execution paths in the graph, determining which nodes are reached by which constraints. This algorithm is more convenient for actually listing the sequences of nodes reached by a constraint than are the flow analysis algorithms usually described in compiler texts (which would tell what constraints reach a node, but not necessarily by what paths). In the following description flow graph nodes are assumed to represent individual source statements, although the algorithm can be easily adapted to graphs in which nodes represent basic blocks. Every flow graph is assumed to have a unique *start node* representing the place at which execution of the program begins.

Algorithm and Basic Definitions

A constrained interval is considered to be a sequence of nodes, $\langle n_1, \ldots, n_k \rangle$, in which n_1 represents a statement that establishes a constraint that affects the rest of the sequence. It is assumed that execution can proceed through the sequence in order, i.e., that there is an edge from n_i to n_{i+1} for all i. This informal description is made precise in the following definitions:

Definition 1: A *constrained interval* in a flow graph is a sequence of nodes $\langle n_1, \ldots, n_k \rangle$ with the following properties:

- n_1 is an opening node (see below).
- n_k matches n_1 (see below).
- No n_i, $i < k$, closes n_1 (see below).
- An edge $n_i \rightarrow n_{i+1}$ exists for all i between 1 and $k-1$.
- n_1 is reachable from the start node.

Definition 2: A flow graph node is an *opening* node if and only if it represents a statement whose execution marks the beginning of a sequence of statements that may be subject to some constraint.

Definition 3: An opening node o is *matched* by node m if and only if execution of (the statement represented by) m marks the end of a sequence of statements subject to some constraint whose beginning is marked by o. Nodes o and m must have distinct positions in the execution sequence, i.e., a node cannot immediately match itself to yield a constrained interval of length one. The constraint(s) to which the sequence is subject must be deducible from o and m. A node that is not opening has no matches.

Definition 4: An opening node o is *closed* by node c if and only if the semantics of the statement represented by c are such that no execution sequence $\langle o, \ldots, c, \ldots m \rangle$ in which o, c, and m have distinct positions and m matches o can exist. A node that is not an opening node is never closed.

Definitions 2, 3, and 4 are necessarily vague about the exact kinds of statement that give rise to opening, closing, and matching nodes, because the statements that serve these roles will vary from language to language. The following example, however, should make the ideas clearer for the constructs discussed in this paper:

```
(1)  timepoint t
     while some_condition
(2)       t'elapsed_time := 100 microseconds
(3)       timepoint t
     end while
(4)  t'elapsed_time :≤ 200 microseconds
```

The flow graph corresponding to this code fragment is shown in Figure 4. Nodes in the flow graph are numbered to correspond to the statements that they represent. Node 1 is an opening node because it marks a point relative to which later references to *t'elapsed_time* may measure time. Since node 2 constrains *t'elapsed_time* and can be reached from node 1 without passing through any nodes that close 1, node 2 matches node 1. Node 3 redeclares the time point declared by node 1, and so closes node 1 (i.e., any references to *t'elapsed_time* that are executed after node 3 is executed will be relative to node 3, not to node 1). Node 3 is also an opening node. Since node 2 is reachable from node 3 via the loop, node 2 matches node 3 (in addition to matching node 1). By the same reasoning that explained why

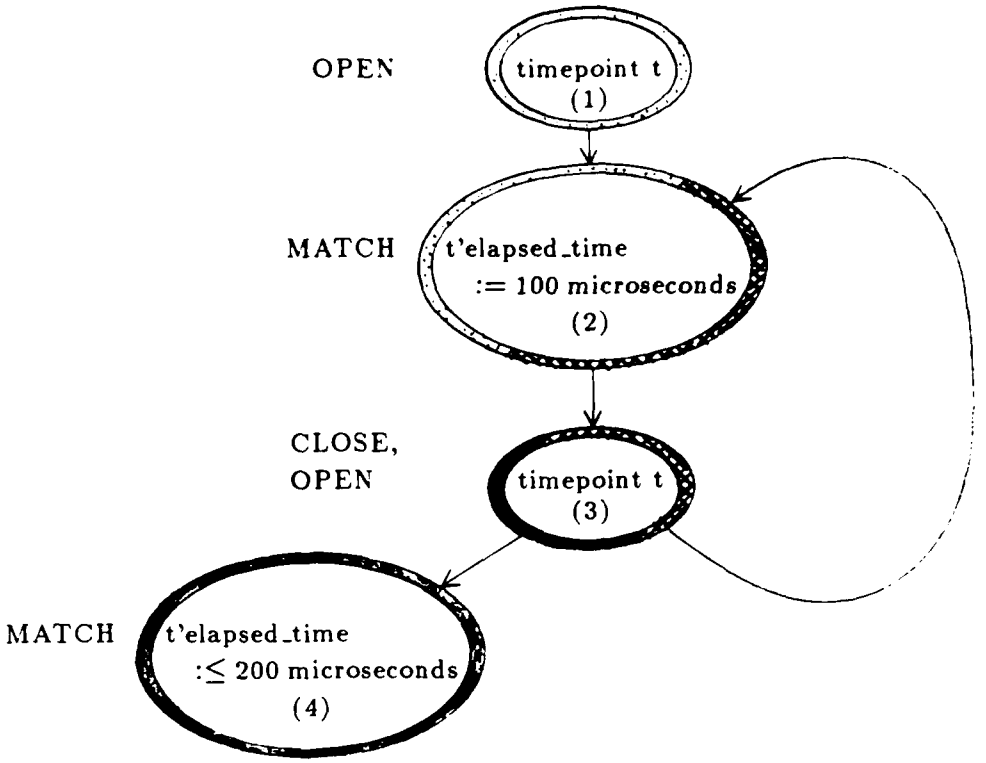


Figure 4. Opening, Closing, and Matching Nodes

node 3 closes node 1, node 3 also closes itself. Note, however, that since a node cannot immediately close itself this statement only means that intervals of the form (node 3, node 2, node 3, ...) cannot exist — the sequence (node 3, node 2) is still a constrained interval. Finally, node 4 matches node 3. Node 4 does not match node 1, since no path from node 1 can reach node 4 without passing through a node that closes node 1. Note that node 4 does not close node 3, even though there are no paths past node 4 that end in a match for node 3. The definition of closing is based on the meaning of statements, not the structure of graphs; so since a constraint on a time point's "elapsed_time" does not preclude later constraints on the same attribute, node 4 does not close node 3. The above analysis indicates that there are three constrained intervals in Figure 4. The Figure distinguishes these intervals by the shading of node borders. Note that several nodes belong to more than one interval.

Another useful concept is that of an *interval prefix*, i.e., a sequence of nodes that could be a prefix of a constrained interval. Formally, an interval prefix has the following features:

Definition 5: An *interval prefix* in a flow graph is a sequence of nodes $\langle n_1, \ldots, n_k \rangle$ such that either $k = 0$ or the following hold:

- n_1 is an opening node.
- No n_i, $i < k$, closes n_1.
- An edge $n_i \rightarrow n_{i+1}$ exists for all i between 1 and $k - 1$.

- n_1 is reachable from the start node.

The definitions given so far raise problems involving cycles in flow graphs. First, if the path defining a constrained interval contains a cycle, then there are an infinite number of other constrained intervals that differ from the first only in how many copies of the cycle they include. Obviously no interval detection algorithm can return all of these intervals. Second, the significance of cycles for constraint satisfaction depends very much on the kinds of constraint involved. A cycle in the flow graph is largely irrelevant to constraints that do not depend on the dynamic behavior of the circuit, for example a constraint on layout area. On the other hand, constraints on such attributes as time generally cannot be satisfied at all over an interval containing a cycle. For these reasons the algorithm for finding constrained intervals is defined only for *cycle-free* constrained intervals. Cycle-free interval prefixes can also be defined. The precise definition of both terms is as follows:

Definition 6: A constrained interval or interval prefix $\langle n_1, \ldots, n_k \rangle$ is *cycle-free* if and only if $n_i \neq n_j$ for all $1 \leq i < j \leq k$, except that n_1 may equal n_k.

The term "cycle-free" may seem something of a misnomer in the above definition, since a cycle-free interval may actually be a single big cycle. The purpose of the definition, however, is to distinguish those intervals that cause the problems discussed above from those that do not. Intervals in which the first and last node are identical generally correspond to meaningful constraints (for instance, a loop in which each iteration must meet some timing constraint), and so are accepted by the definition. The real problems (particularly infinite numbers of intervals) occur when intervals longer than a single copy of a cycle exist, and these intervals are excluded by the definition. In other words, the proper meaning of "cycle-free" is "free of cycles that cause problems".

Using these definitions, the algorithm for finding constrained intervals in a flow graph (Intervalize) can be given. Most of the work of Intervalize is done by procedure *trace*, which carries out the basic depth-first traversal. The central point of this traversal is to locate matches for the *open* argument to *trace*, and to record the path from *open* to each match in the global variable *intervals*. The *node* argument to *trace* is the current potential match; *path* is a sequence of nodes describing the path by which *node* was reached from *open*. The operator "·" is used to indicate concatenation of a scalar onto a sequence. Intervalize returns a set of sequences of nodes, with each sequence being a constrained interval in the flow graph. Note that all opening nodes must be identified prior to running Intervalize. This is easily done by one or more initial passes over the flow graph

Algorithm 1 (Intervalize):
Input: A flow graph for a behavioral specification.
Output: A set of constrained intervals.
Precondition: All opening nodes in the flow graph have been identified.

```
intervals ← ∅
for each opening node n reachable from the start node in the flow graph
   call trace(n, n, ⟨⟩)
return intervals

trace(node, open, path)
     if node matches open
       intervals ← intervals ∪ {path · node}
     if node appears in path
       Cycle detected — handling depends on specific application.
     else
       if node does not close open
         for each successor s of node
           if s = open or s does not appear in path · node
             call trace(s, open, path·node)
```

Correctness

The obvious criterion for correctness of Intervalize is that it find all the cycle-free constrained intervals in a flow graph and nothing else. The reasons for excluding constrained intervals containing cycles were mentioned earlier. Note, however, that the proper point for extending Intervalize with application-specific cycle handlers is clearly marked above.

The first step in the correctness proof is to prove a simple Lemma that explains the role of the *open* argument to *trace*. This Lemma just states that the *open* argument is always the opening node of the interval prefix currently being explored by *trace*. This fact is easily seen to be true by inspecting Intervalize, but so many later steps rely on it that it is worth proving formally.

Lemma 1: The first node of the sequence *path* · *node* in any invocation of *trace* is that invocation's *open* argument.

Proof. The proof is by induction on the length of *path* · *node*. If this sequence only contains one node, then the corresponding invocation of *trace* must have been made from the top-level loop of Intervalize (since no other invocations have empty *path* arguments). In this case *node* and *open* are identical and *path* is empty. Thus *path* · *node* = ⟨*node*⟩ and since *node* = *open* the Lemma holds. Assume that the Lemma holds for *path* · *node* of length k, $k \geq 1$, and let $p = \langle n_1, \ldots, n_k, n_{k+1} \rangle$ be a *path* · *node* sequence of length $k+1$. Since $k \geq 1$, the invocation of *trace* corresponding to p must be one of the recursive ones. It's caller must have had *path* · *node* = $\langle n_1, \ldots, n_k \rangle$, which is of length k, so by the induction hypothesis *open* = n_1 in that invocation. Since recursive calls on *trace* do not change *open*, the invocation corresponding to p must also have *open* = n_1. □

The next step in the correctness proof is to show that Intervalize always terminates. This is done in two steps. The first step (Lemma 2) consists of showing that *trace* is always working on some cycle-free interval prefix. The second step (Theorem 1) involves showing that the only way Intervalize can fail to terminate is if *trace* goes into an infinitely deep recursion. Such a recursion would require an infinitely long cycle-free interval prefix for *trace* to work on, which is impossible as long as the flow graph given to Intervalize is finite. The formal statement of these results follows:

Lemma 2: The *path*·*node* sequence in any invocation of *trace* is a cycle-free interval prefix of the flow graph given to Intervalize.

Proof. Let $p = \langle n_1, \ldots, n_k \rangle$ be a *path* · *node* sequence in *trace*. The proof is by induction on k, the length of p. If $k = 1$ then p must be associated with an invocation of *trace* that was called from the top-level loop in Intervalize. By the construction of this loop, the only node in p is an opening node reachable from the start node, so the first and fourth conditions for being an interval prefix hold. The second (no node but the last closes the first) and third (there is an edge between every pair of adjacent nodes in the sequence) hold trivially. Thus p is an interval prefix, and must be cycle-free simply because it does not contain enough nodes to have a cycle. Assume now that the lemma holds for *path* · *node* sequences of length l, $l \geq 1$, and consider $p = \langle n_1, \ldots, n_l, n_{l+1} \rangle$ of length $l+1$. Since $l \geq 1$, p must be associated with one of the recursive invocations of *trace*. Thus p can be written as $p' \cdot n_{l+1}$, where p' was the *path* · *node* sequence in the previous invocation of *trace* and n_{l+1} is some successor of n_l. Since p' is of length l, it is a cycle-free interval prefix by the induction hypothesis. Since n_{l+1} is a successor of n_l, and p' is an interval prefix, there are edges between every pair of nodes in p (third condition for being an interval prefix). Also since p' is an interval prefix, n_1 is an opening node reachable from the start node (first and fourth conditions). Finally, no n_i, $i < l+1$ can close n_1 (second condition), since p' is an interval prefix and if n_l closed n_1 then the recursive call could not have been made (by the "if *node* does not close *open*" test and Lemma 1). Thus p is itself an interval prefix. Since p' is cycle-free, the only ways in which p can fail to be cycle-free are if $n_l = n_1$ or if $n_{l+1} = n_i$, $1 < i < l+1$. In either case, however, the recursive call could not have been made, either because of the "if *node* appears in *path*" test or because of the "if s = *open* or s does not appear in *path* · *node*" test. Thus p is cycle-free as well as being an interval prefix. □

Theorem 1: Intervalize terminates when applied to any finite flow graph.

Proof. Since there are a finite number of nodes in the flow graph, the loop in the main body of Intervalize repeats only a finite number of times. The only way for Intervalize not to terminate is thus for one of this loop's calls on *trace* to fail to terminate. This in turn can only happen if *trace* makes an infinitely deep series of recursive calls on itself. Note that in every recursive call, the called invocation of *trace* must have a *path* · *node* sequence that is one node longer than its caller's *path* · *node* sequence. Since these sequences are always cycle-free interval prefixes

(by Lemma 2), an infinitely deep series of recursive calls implies that there is an infinitely long cycle-free interval prefix in the flow graph. However, the definition of "cycle-free" implies that no cycle-free interval prefix can contain more more than $n + 1$ nodes, where n is the number of nodes in the flow graph. Since n is finite, infinite recursions are impossible, and Intervalize must always terminate. □

The last step in the correctness proof is to show that when Intervalize stops, *intervals* contains exactly the set of cycle-free constrained intervals from the input flow graph. The proof begins with Lemma 2 and the fact that every non-empty cycle-free interval prefix is eventually available to *trace* (Lemma 3). From these facts it can be shown that only cycle-free constrained intervals are placed in *intervals*, and that all cycle-free constrained intervals are eventually placed there.

Lemma 3: Every cycle-free interval prefix of length 1 or greater in the flow graph given to Intervalize is equal to $path \cdot node$ in some invocation of *trace*.

Proof. Assume that there is some cycle-free interval prefix $p = \langle n_1, \ldots, n_k \rangle$, $k \geq 1$, such that there is no invocation of *trace* with $p = path \cdot node$. Let $p' = \langle n_1, \ldots, n_i \rangle$ be the longest prefix of p such that there is an invocation of *trace* with $path = \langle n_1, \ldots, n_{i-1} \rangle$ and $node = n_i$ (i.e., p' is equal to $path \cdot node$ in some invocation of *trace*). Note that p' must be a proper prefix of p by the assumption that $p \neq path \cdot node$ in any invocation of *trace*. p' must exist and be non-empty, because n_1 is an opening node (by the definition of interval prefix) and p is non-empty. The top-level loop of Intervalize ensures that an invocation of the form $trace(n, n, \langle\rangle)$ is made for every opening node n, so one of these invocations will have $path \cdot node = \langle n_1 \rangle$. This sequence is therefore the shortest possible p'. Consider the invocation of *trace* with $path = \langle n_1, \ldots, n_{i-1} \rangle$ and $node = n_i$. Clearly n_i is not an element of $\langle n_1, \ldots, n_{i-1} \rangle$, nor is n_{i+1} an element of $\langle n_2, \ldots, n_i \rangle$, because p' is a proper prefix of p and p is cycle-free. Also, n_i cannot close n_1, because p' is a proper prefix of p and p is an interval prefix. For the same reason, there is an edge from n_i to n_{i+1}. Thus the loop over successors of n_i will be entered (Lemma 1) and there will be an invocation of the form $trace(n_{i+1}, n_1, \langle n_1, \ldots, n_i \rangle)$. This invocation will have $path \cdot node$ equal to a longer prefix of p than p', which contradicts the definition of p'. Thus the assumption that p exists at all must be invalid, and so the Lemma is proved. □

Theorem 2: Upon termination of Intervalize, *intervals* contains all and only the cycle-free constrained intervals from the input flow graph.

Proof. The proof that all cycle-free constrained intervals eventually become members of *intervals* is as follows: Let p be any cycle-free constrained interval from the flow graph given to Intervalize. By definition, p is also a non-empty cycle-free interval prefix. Thus, by Lemma 3, there is some invocation of *trace* in which $path \cdot node = p$. Since p is a constrained interval, *node* matches *open* in this invocation (relying on Lemma 1 again). Furthermore, since p is cycle-free, *node* does not appear in *path*, except possibly as the first element. Thus the test guarding the "$intervals \leftarrow intervals \cup \{path \cdot node\}$" statement will succeed, and p will be added to *intervals* if it is not already there. The proof that only

cycle-free constrained intervals are added to *intervals* follows from Lemma 2. This Lemma ensures that the only sequences that can be added to *intervals* by the "*intervals* ← *intervals* ∪ {*path* · *node*}" statement are cycle-free interval prefixes. The test guarding this statement ensures that the last node of these interval prefixes matches the first, so in fact they are cycle-free constrained intervals. Since *intervals* starts out empty, and is only changed by the "*intervals* ← *intervals* ∪ {*path* · *node*}" statement, *intervals* can never contain anything but cycle-free constrained intervals. □

Execution Time

The following paragraphs discuss the running time of Intervalize. Note that since Intervalize is basically just a series of depth-first searches of a flow graph, it can take no more than $O(on)$ time, where o is the number of opening nodes in the graph and n is the total number of nodes of any kind. However, only part of the graph is examined for each opening node, so this bound is rather loose. A tighter bound of $\Theta(p)$, where p is the number of cycle-free interval prefixes in a graph, is derived below. Intuitively this time seems good, but it remains unknown whether it is truly optimal.

Note that the running time of Intervalize can be assumed to be determined by the total number of calls made on *trace*. This assumption is justified by the fact that the top level of Intervalize consists of essentially nothing but a series of calls on *trace*, and that *trace* itself contains some constant-time processing followed by a loop that either calls *trace* or does nothing on each iteration. It could be argued that the various tests in *trace* to see if some node appears in *path* or *path* · *node* take time proportional to the length of *path*. This problem can be solved by including a marker bit in each node. *Trace* can set this bit whenever a node is added to *path*, and clear the bit while backing out of recursive calls. Testing to see if a node is an element of the current path is then simply a matter of testing to see whether its marker is set, which can be done in constant time.

Using the above assumption, an expression for the running time of Intervalize is derived by counting the number of times *trace* is called. Lemma 3 provides a lower bound, since it says that *trace* must be called at least as many times as there are cycle-free interval prefixes in the flow graph on which Intervalize is working. To get an upper bound, a similar lemma (Lemma 4) is proved below, stating that *trace* is called no more than once for each cycle-free interval prefix. Together, these two lemmas establish the number of cycle-free interval prefixes in a flow graph as a tight bound on the number of calls on *trace*, and hence on the running time of Intervalize.

Lemma 4: When running Intervalize on any flow graph, no sequence of nodes appears as *path* · *node* in more than one invocation of *trace*.

Proof. The proof is by induction on the length of the sequence. Sequences of length 1 only appear in the invocations made from the top-level loop of Intervalize, and this loop iterates only once for each opening node. Thus no two invocations from this loop can have the same *path* · *node*, and so the lemma holds. Assume that

the lemma holds for sequences of length k, $k \geq 1$. Let $p = \langle n_1, \ldots, n_k, n_{k+1} \rangle$ be a sequence of length $k+1$ that appears as *path* · *node* in two different invocations of *trace*. Since $k \geq 1$, both of these invocations must be recursive ones. The caller of each of these invocations must have had *path* · *node* $= \langle n_1, \ldots, n_k \rangle$, which is of length k. Thus both invocations must have the same caller — otherwise there would be two different invocations with identical length k *path* · *node* sequences, which is impossible by the induction hypothesis. However, recursive calls on *trace* are only made from a loop that has one iteration per successor of the current *node*, and so a single invocation of *trace* cannot make two distinct recursive calls with the same *path* · *node* sequence (assuming that multiple edges between nodes do not exist). Thus no two invocations of *trace* can have identical length $k+1$ *path* · *node* sequences. □

Theorem 3: The execution time of Intervalize on a flow graph containing p non-empty cycle-free interval prefixes is $\Theta(p)$.

Proof. Lemma 3 says that there must be at least p calls on *trace*, so execution time must be $\Omega(p)$. Since the *path* · *node* sequence in any invocation of *trace* is a cycle-free interval prefix (Lemma 2), Lemma 4 means that there are no more than p calls on *trace*. It follows from this fact that the execution time of *trace* is $O(p)$. Since the execution time of *trace* is both $O(p)$ and $\Omega(p)$ it must also be $\Theta(p)$. □

An execution time of $\Theta(p)$ for finding constrained intervals seems pretty good, since one might expect that each interval prefix has to be examined at least once to see if it is actually a constrained interval. However, I have not proven that this execution time is optimal, and in fact faster algorithms can be imagined. For example, since each node in a flow graph can be a component of several interval prefixes, one might believe that algorithms exist that only look at each node once (Intervalize looks at each node as many times as there are cycle-free interval prefixes containing it). Thus, although Intervalize should be entirely suitable for practical use, it remains an open question whether asymptotically faster algorithms exist.

Summary

The important points of this section have been made via a series of technical definitions and proofs. To summarize, the section began with a series of definitions that precisely stated what sections of specifications are subject to constraints. These definitions were used in an algorithm (Intervalize) for finding code fragments that are subject to constraints. A series of theorems and lemmas led to two important conclusions about the correctness of Intervalize, namely that it always stops and that when it stops it has found exactly the "interesting" constrained fragments of its input. Finally, it was shown that the running time of Intervalize is tightly bounded by the number of potential constrained fragments in its input. It was conjectured that this running time is "good", but it is not yet known how much better one could do.

4 Experience

I first studied the inclusion of physical constraints in behavioral specifications during a project aimed at finding ways of stating physical constraints in the SILC™ [4] language. The basic ideas of attributes and assignment-like constraints were developed during this project, as were some constraint types and attributes specific to SILC. These ideas were first tested in descriptions of a few small telecommunications circuits. They were then more extensively tested in a detailed description of a floppy-disk controller. This circuit was chosen as a test because its complexity is typical of modern VLSI designs, and because timing constraints are a crucial part of its behavior. The resulting description contained some 1400 lines of extended SILC code, roughly 70 of which were for the specification of constraints. All the constraints were timing constraints. The low ratio of constraint statements to others suggests that the mechanisms proposed here are very expressive despite their simplicity. Constraints on "Hold_Time" and "Ignore_Time" attributes of inputs and outputs turn out to be particularly powerful, because a single constraint statement can establish all the timing for a number of I/O operations executed over a long period of time.

It is difficult to compare the floppy disk controller example to the telecommunications circuits that were also studied because of the immense difference in size. However, it does appear that the use of only timing constraints in the disk controller is not unusual. Most of the telecommunications examples required two or three timing constraints, but no more than one or two constraints on all other attributes. These non-timing constraints were always very stylized minimizations of things like power consumption or layout area. These observations, although limited, reinforce the intuition that the very simple kinds of constraint suggested here really are adequate for practical circuit design.

The ideas developed during the GTE Laboratories project have been further refined in my present work on RASP. A RASP source language (RASP-SL), which provides many of the features described in this paper, has been designed. The constraint-oriented features of RASP-SL are summarized in Table 1. Although a few small specifications have been written in RASP-SL (for example, the serial transmitter in Figure 2 is adapted from a RASP-SL specification), the RASP project is mainly intended to show that automatic, constraint-sensitive synthesis is practical. Most of the effort has therefore been on identifying and testing the implementation strategies described in the previous section. Work is now in progress to implement Intervalize and other front-end analyses in Common Lisp on various workstations (notably Texas Instruments' Explorer™).

SILC is a trademark of Silc Technologies, Inc.

Explorer is a trademark of Texas Instruments, Inc.

Program Unit Attributes:
  Power
  Area
  Clock Period

Variable Attributes:
  Hold Time
  Ignore Time

Time Point Attributes:
  Elapsed Time

Constraints:
  $\leq, \geq, =$
  Maximize, Minimize

Miscellaneous:
  Time points
  "unconstrain"

Table 1. Attributes and Constraints in RASP's Source Language

5 Summary

I have proposed features that can be added to any behavioral description language in order to describe physical resource constraints. In many cases, especially those involving time, correct circuit behaviors cannot be fully described without such constraints. These features have been embedded in several languages, including one (SILC) that did not originally include them, and one (RASP-SL) that was designed from the start with them in mind. Large, realistic devices have been described using these mechanisms, indicating that they are powerful enough for practical use. Finally, methods for automatically generating circuits from descriptions containing constraints have been suggested. Research is in progress to find out how well these methods work.

The work described in this paper is unique in providing a comprehensive way of describing the physical constraints on a circuit's behavior in a form that can easily be combined with a functional description. The ability to constrain both execution and data representation is an important part of this work. The power of constrainable attributes has been demonstrated on realistic circuit descriptions, and work on building translators for them is in progress. These ideas should be an important extension to existing methods of behavioral description as automatic circuit design becomes a reality.

Acknowledgements

The work described here could never have been done without the encouragement and support of the SILC group at GTE Laboratories. In particular, Jeff

Fox, John Blank, Tim Blackman and Martin Resnick have been extraordinarily helpful. Mandayam Srinivas and Mark Fulk at the University of Rochester made helpful comments on the presentation of Intervalize and the proofs associated with it. Needless to say, the format of this paper and the opinions expressed therein are solely the responsibility of the author, and are not necessarily endorsed by any of the aforementioned people or organizations.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools*. Reading, Ma.: Addison-Wesley Publishing Co. 1986.

[2] D. Baldwin. "A Model for Automatic Design of Digital Circuits". Technical Report number 188, Department of Computer Science, University of Rochester. July 1986.

[3] M. Barbacci. "Instruction Set Processor Specifications (ISPS): The Notation and its Applications". *IEEE Transactions on Computers*, Jan. 1981, (**C-30:1**). pp. 24–40.

[4] T. Blackman, J. Fox, and C. Rosebrugh. "The SILC Silicon Compiler: Language and Features". Proceedings of the 22nd ACM/IEEE Design Automation Conference, June 1985, pp. 232–237.

[5] A. Borning. "Thinglab — A Constraint-Oriented Simulation Laboratory". Technical Report number SSL–79–3, Xerox Palo Alto Research Center. July 1979.

[6] R. Camposano and A. Kunzmann. "Considering Timing Constraints in Synthesis from a Behavioural Description". Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers, Oct. 1986. pp. 6–9.

[7] J. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Transactions on Computers*, July 1981, (**C-30:7**), pp. 478–490.

[8] J. Gosling. "Algebraic Constraints". Department of Computer Science, Carnegie-Mellon University, Technical Report number CMU-CS-83-132. May 1983.

[9] L. Hafer and A. Parker. "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic". Proceedings of the 18th Design Automation Conference, June 1981. pp. 846–853.

[10] Intermetrics, Inc. *VHDL User's Manual*, vol. 2: *User's Reference Guide*. Bethesda, Md: 1985.

[11] D. Landskov *et al.* "Local Microcode Compaction Techniques". *ACM Computing Surveys*, Sept. 1980 (**12:3**), pp. 261–294.

[12] P. Marwedel. "The MIMOLA Design System: Tools for the Design of Digital Processors". Proceedings of the ACM IEEE 21st Design Automation Conference, June 1984, pp. 587–593.

[13] J. Nestor and D. Thomas. "Behavioral Synthesis with Interfaces". Digest of Technical Papers, IEEE International Conference on Computer Aided Design (ICCAD-86), Nov. 1986. pp. 112–115.

[14] N. Park. "Sehwa: A Program for Synthesis of Pipelines". Proceedings of the 23rd ACM/IEEE Design Automation Conference, June 1986. pp. 454–460.

[15] P. Piloty *et al.* *CONLAN Report.* Berlin: Springer-Verlag (*Lecture Notes in Computer Science* 151). 1983.

[16] D. Siewiorek and M. Barbacci. "The CMU RT-CAD System -- An Innovative Approach to Computer Aided Design". Proceedings of the 1976 National Computer Conference. pp. 643–655.

[17] I. Sutherland. "SKETCHPAD: A Man-Machine Graphical Communication System". Technical Report number 296, MIT Lincoln Laboratory, Jan. 1963.

END

DATE

FILMED

MARCH

1988

DTIC